Section 12: Networking, 2PC, RPC

CS 162

November 20, 2020

Contents

1 Vocabulary

- **TCP** - Transmission Control Protocol (TCP) is a common L4 (transport layer) protocol that guarantees reliable in-order delivery. In-order delivery is accomplished through the use of sequence numbers attached to every data packet, and reliable delivery is accomplished through the use of ACKs (acknowledgements).
- **Primary/Secondary** or **Coordinator/Worker**. A scheme for separation of responsibilities. In this scheme, there is typically a single active primary and one or more secondaries. The primary is typically responsible for being the source of truth and directing operations towards the secondaries.
- **Failover** A fault tolerance procedure invoked when a component fails. Typically this involves switching over to, or promoting a secondary.
- **2PC** - Two Phase Commit (2PC) is an algorithm that coordinates transactions between one coordinator and many workers. Transactions that change the state of the worker are considered 2PC transactions and must be logged and tracked according to the 2PC algorithm. 2PC ensures atomicity and durability by ensuring that a write happens across ALL replicas or NONE of them. The replication factor indicates how many different workers a particular entry is copied among. The sequence of message passing is as follows:

```
for every worker replica and an ACTION from the coordinator,
origin [MESSAGE] -> dest :
---
COORDINATOR [VOTE-REQUEST(ACTION)] -> WORKER
WORKER [VOTE-ABORT/COMMIT] -> COORDINATOR
COORDINATOR [GLOBAL-COMMIT/ABORT] -> WORKER
WORKER [ACK] -> COORDINATOR
```

If at least one worker votes to abort, the coordinator sends a GLOBAL-ABORT. If all worker vote to commit, the coordinator sends GLOBAL-COMMIT. Whenever a coordinator receives a response from a worker, it may assume that the previous request has been recognized and committed to log and is therefore fault tolerant. (If the coordinator receives a VOTE, the coordinator can assume that the worker has logged the action it is voting on. If the coordinator receives an ACK for a GLOBAL-COMMIT, it can assume that action has been executed, saved, and logged such that it will remain consistent even if the worker dies and rebuilds.)

- **Endianness** - The order in which the bytes are stored for integers that are larger than a byte. The two variants are big-endian where byte the most significant byte is at the lowest address and the least significant byte is at the highest address and little-endian where the least significant byte is at the lowest address and the most significant byte is at the highest address. The network is defined to be big-endian whereas most of your machines are likely little-endian.
- `uint32_t htonl(uint32_t hostlong)` - Function to abstract away endianness by converting from the host endianness to the network endianness.
- `uint32_t ntohl(uint32_t netlong)` - Function to abstract away endianness by converting from the network endianness to the host endianness.
- **RPC** - Remote procedures calls are a technique for distributed computation through a client server model. This process effectively calls a procedure on a possibly remote server from a client by wrapping communication over the network with wrapper stub functions. This six steps are:
    1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
    2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshaling.
    3. The client's local operating system sends the message from the client machine to the server machine.
    4. The local operating system on the server machine passes the incoming packets to the server stub.
    5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshaling.
    6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction

2 Networking

a) (True/False) IPv4 can support up to 2^{64} different hosts.

b) (True/False) Port numbers are in the IP header.

c) (True/False) UDP has a built in abstraction for sending packets in an in order fashion.

d) (True/False) TCP provide a reliable and ordered byte stream abstraction to networking.

e) (True/False) TCP attempts to solve the congestion control problem by adjusting the sending window when packets are dropped.

f) In TCP, how do we achieve logically ordered packets despite the out of order delivery of the physical reality? What field of the TCP packet is used for this?

g) Describe how a client opens a TCP connection with the server. Elaborate on how the sequence number is initially chosen.

h) Describe the semantics of the acknowledgement field and also the window field in a TCP ack.

i) List the 5 layers specified in the TCP/IP model. Layering adds modularity to the internet and allows innovation to happen at all layers largely in parallel. What is the function of each layer?

j) The end to end principle is one of the most famed design principles in all of engineering. It argues that functionality should **only** be placed in the network if certain conditions are met. Otherwise, they should be implemented in the end hosts. These conditions are:

- Only If Sufficient: Don't implement a function in the network unless it can be completely implemented at this level.
- Only If Necessary: Don't implement anything in the network that can be implemented correctly by the hosts.
- Only If Useful: If hosts can implement functionality correctly, implement it in the network only as a performance enhancement.

Take for example the concept of reliability: making all efforts to ensure that a packet sent is not lost or corrupted and is indeed received by the other end. Using each of the three criteria, argue if reliability should be implemented in the network.

i) Only If Sufficient

ii) Only If Necessary

iii) Only If Useful

3 Two-Phase Commit

Quorum consensus is able to provide weak consistency. For this section, assume the DHT backs each node with multiple replicas. Further, assume that these machines use a two phase commit protocol to commit information in a strongly consistent manner.

1. 2PC requires a single coordinator and at least one worker. By default, all replicas can be workers. How should the coordinator be picked?

2. Briefly describe the messages that the **coordinator** will send and receive in response to a PUT. Also describe when and what would be logged.

3. Briefly describe the messages that a **worker** will send and receive.

4. Under the current model, multiple PUT queries could be sent to separate coordinator machines. Propose set of worker routines which can handle this. In particular, consider the case in which 2 coordinators receive PUT queries for **the same key but different values**. The state of the system should remain consistent.

4 RPC

To explore the process for performing RPC we will consider implement the server end for two procedures. We want to implement the server side of an RPC version of the following code

```
// Returns the ith prime number (0 indexed)
uint32_t ith_prime (uint32_t i);

// Returns 1 if x and y are coprime, otherwise 0.
uint32_t is_coprime(uint32_t x, uint32_t y);
```

Assume the server has already implemented `ith_prime` and `is_coprime` locally.

4.1 Selecting Functions

As a first step we receive data from the client. How do we decide which procedure we are executing? Provide a sample header file addition that could be used to indicate this.

4.2 Reading Arguments

When examining the arguments for your two functions you notice that the arguments require either 8 or 4 bytes, so you believe you can handle either case by attempting to read 8 bytes using the code below.

```
// Assume dest has enough space allocated
void read_args (int sock_fd, char *dest) {
    int byte_len = 0;
    int read_bytes = 0;
    while ((read_bytes = read (fd, dest, 8 - byte_len)) > 0) {
        byte_len += read_bytes
    }
}
```

However when you implement it you notice that for some inputs your server appears to be stuck? Why might this be happening and for which inputs could this happen?

4.3 Handling RPC

Realizing your previous solution was insufficient you decide to implement a slightly more complicated protocol. You settle on the following steps for the client:

1. The client sends an identifier of the function it wants as an integer (0 for `ith_prime`, 1 for `is_coprime`).
2. The client sends all the bytes for all the arguments.

The server then takes the following steps:

1. The server reads the identifier.
2. The server uses the identifier to allocate memory and set the read size.
3. The server reads the remaining arguments.

Complete the following function to implement the server side of handling data. You may find `ntohl` useful.

```
// Function to implement the server side of the protocol.
// Returns whether or not it was successful and closes the socket when finished.
// Assume get_sizes loads all our sizes based on id and call_server_stub selects
// our host function
void receive_rpc (int sock_fd) {
    uint32_t id;
    char *args;
    size_t arg_bytes;
    char *rets;
    size_t ret_bytes;
    int bytes_read = 0;
    int bytes_written = 0;
    int curr_read = 0;
    int curr_write = 0;
    while ((_____________________________________________________________
                    ____________________________________________________) > 0) {
        bytes_read += curr_read;
    }
    ____________________________________________________________________________;
    // Allocates sizes based on our id (not a real library function)
    get_sizes (id, &args, &arg_bytes, &rets, &ret_bytes);
    bytes_read = 0;
    while ((_____________________________________________________________
                    ____________________________________________________) > 0) {
        bytes_read += curr_read;
```

```
    }
    // Calls the appropriate server stub based on id (not a real library function)
    call_server_stub (id, args, arg_bytes, rets, ret_bytes)
    while ((____________________________________________________________
                  ____________________________________________________) > 0) {
        bytes_written += curr_write;
    }
    ____________________________________________________________________________;
}
```

4.4 Call Stubs

Finally we want to implement the call stubs, which are function wrappers between each individual function we support and the generic RPC library. For example these are the client stubs for our functions:

```
// addrs is used for setting up our socket connection
uint32_t ith_prime_cstub (struct addrinfo *addrs, uint32_t i) {
    uint32_t prime_val;
    i = htonl(i);
    call_rpc (addrs, RPC_PRIME, (char *)&i, 1, (char *) &prime_val, 1);
    return ntohl(prime_val);
}
uint32_t is_coprime_cstub (struct addrinfo *addrs, uint32_t x, uint32_t y) {
    uint32_t coprime_val;
    uint32_t args[2] = {htonl(x), htonl(y)};
    call_rpc (addrs, RPC_COPRIME, (char *) args, 2, (char *) &coprime_val, 1);
    return ntohl(coprime_val);
}
```

`call_rpc` is our generic rpc handler that our stubs provide an abstraction around. Notice that we marshal arguments with `htonl()` and unmarshal them with `ntohl()`.

After the raw data is processed we need to perform the actual computation and return the result to the client. To do this we introduce a stub procedure which unpacks our arguments, calls the procedure to execute on the server, and finally packs the return results to give to the transport layer.

Implement `ith_prime_sstub` and `is_coprime_sstub` which should unmarshal the arguments, call the implementation functions, and finally marshal the return data.

```
// Both functions take in args in network order and should place
// return value(s) in rets in network order
void ith_prime_sstub (char *args, char *rets) {
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
}
void is_coprime_sstub (char *args, char *rets) {
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
```

```
    ------------------------------------------------------------------
    ------------------------------------------------------------------
    ------------------------------------------------------------------
    ------------------------------------------------------------------
}
```

4.5 Handling failure

What are some ways a remote procedure call can fail? How can we deal with these failures?